C/C++ Contributing Editors

The New C: Inline Functions

Randy Meyers

As if C weren't fast enough already, C99 supports inline functions. Faster code, anyone?

When C was first invented, the **register** keyword was a good idea. Back in those days, many compilers for most languages ran in 64 KB or less of memory. Even on mainframes, optimizing compilers (and optimizing compilers were unusual) for large languages like PL/I might run in only 256 KB of memory. The algorithms for register allocation were somewhat new and had a tendency to dramatically increase the effort to write compilers as well as the memory and execution time that compilers required. Due to the tight memory constraints, compilers tended to process each source statement in isolation from other statements. Such compilers would do all of the work of compiling a statement, from parsing to code generation, before moving to the next statement.

That sort of compiler organization precludes good register allocation since good register allocation requires analyzing all of the statements in a function before making any decisions. For example, the best register allocation might be to allocate no registers to variables used in the current statement because the statements that follow need the registers more for other purposes. The **register** keyword in C was a great help to such compilers, since it allowed the programmer to tell the compiler something that the compiler might not be able to figure out on its own.

Modern compilers no longer compile a statement at a time. Taking advantage of the megabytes of memory now available, compilers translate the entire source module into an internal representation, which is then repeatedly analyzed in order to make good decisions about code generation. These days, compilers are as good or better than programmers at register allocation. And thus, most modern compilers ignore the **register** keyword. (Actually, the C Standard requires that compilers issue a diagnostic if the address-of operator is applied to a variable declared **register**. Compilers note that a variable was declared **register** only to produce that message.)

The subject of this month's column is the modern equivalent of the **register** keyword: the **inline** keyword allows the programmer to tell the compiler something it might have a hard time figuring out automatically. However, in the future, compilers may be able to do a better job of making inline decisions than programmers. When that happens, the **inline** keyword might be regarded as a quaint reminder of when programmers were forced to worry about details of code generation. Until that happens, programmers should be aware of how to use the new C99 **inline** keyword.

Inline Substitution Optimization

The optimization underlying the **inline** keyword is an inline function call substitution. This optimization is similar in some ways to macro expansion in that the

code for a function is inserted inline at the point a function is called. Given a function:

```
void f(int *x, int y)
{
    *x = 10*y;
}
```

and a call to that function:

```
extern int a, b, c;
void caller1()
{
    a = 10*b;
    f(&c, b);
}
```

The body of that function can be substituted for the call to that function, in effect rewriting the caller as:

```
void caller1()
{  // after inline substitution
    a = 10*b;
    *&c = 10*b;
}
```

However, unlike macro expansion, inline substitution is not textual replacement. The compiler must be very careful to preserve the exact semantics of the function call so that the program cannot tell if the optimization was performed or not. This includes such properties of function calls as the arguments being evaluated exactly once, that variable names in the called function are distinct from the caller, and that the parameters of the called function are distinct variables from the arguments passed. Thus, the rewrite of **caller1()** above is actually an optimized version of the inline substitution that the compiler first performs. The compiler probably first rewrote **caller1()** as:

```
void caller1()
{
    a = 10*b;
    {
      int *_F_x = &c;
      int _F_y = b;
      *_F_x = 10*_F_y;
    }
}
```

Note several things about this rewrite. All parameters **f** became local variables of a new block representing the inline expansion. (The **_F_** prefix added to variable names local to the inline avoids conflicts between the names from the inline substitution and names used in the argument expressions.) Those parameters/local variables were initialized with the values of the arguments when the block is entered. Thus, the arguments are evaluated exactly once. The local variable representing the parameters perfectly capture the semantics of parameters of functions: they act as distinct local variables that if assigned to, do not alter the original arguments "passed" to the function.

Further optimizations performed by a compiler may dramatically simplify the code. For example, a common optimization is for a compiler to recognize that a variable only exists to be a copy of another variable. A similar optimization is to sometimes replace a variable with the expression that gave the variable its last value. Since the function **f** contains no assignments to its parameters, a compiler is likely to optimize **caller1()** into:

```
void caller1()
{
    a = 10*b;
    {
      int *_F_x;
      int _F_y;
      *&c = 10*b;
    }
}
```

Further common optimizations are to eliminate variables that are not used, remove blocks that have no local variables, and to eliminate pairs of indirection operators immediately followed by address-of operators. Thus, after inline substitution and further optimization, **caller1()** performs as if it was written:

```
void caller1()
{
    a = 10*b;
    c = 10*b;
}
```

It is important to realize that the compiler was careful to perform the initial inline substitution in a way that preserved the semantics of a function call and then performed general optimizations, which are done to both code resulting from inlining and code written directly by the programmer, to transform the program in ways that do not alter the results. For example, if **f** is assigned into its parameter **y**, then the local variable for the parameter **y** and its initialization would not have been optimized away. Likewise, if the arguments had been expressions with side effects, the compiler would not have eliminated the local variables for the parameters (except if very special conditions existed). All of the optimizations are careful to preserve the meaning of the function call. Thus, a call to **f** is the same whether an actual call is made or the body of **f** is inline substituted.

Inline Advantages

Inline substitution can pay off in several ways. First, it can eliminate the overhead in doing a function call. When a function is called, the following steps are usually taken:

- Argument values are copied to the stack or special registers.
- A return address is created and stored on the stack or to a register.
- The program branches to the function.
- A stack frame is set up for the local variables of the function.
- After the function finishes, the stack frame is torn down.
- The return address is retrieved.
- A branch is made to the return address.

This overhead can be a sizable percentage of the execution of very small functions. As we have already seen, inlining frequently can eliminate the copies of the arguments. There is no return address or branches since the inline substitution is straight-line code. (On many modern machines, branches are expensive because they tend to disrupt the instruction pipeline.) Even allocating the space for the stack frame may be eliminated or folded into setting up the stack frame for the enclosing block by the optimizer.

Inline substitution allows the optimizer to do a better job. For example, the expression **10*b** is a common sub-expression that occurs both in the original body of **caller1()** and the inline substituted code. After inlining, the optimizer can recognize that **10*b** has the same value in both places and compute that expression only once and use it twice. Likewise, if the call **f(&c, 10)** was made, the compiler could perform the arithmetic in the assignment to **c** at compile time.

Inline substitution can also aid register allocation. First, by analyzing both the caller and the inlined body of a called function, the register allocator can do a better job. Second, many calling standards set aside some number of temporary registers that are not saved and restored when calling a function. These registers are used to hold intermediate results and common sub-expressions, but their values are considered to have been lost if a function call occurs since the called function might have used those temporary registers for its own purposes. If a function call is inlined, then there is no actual function call, and the compiler can determine whether any temporary register actually was reused for another purpose. This allows the compiler to manage the temporary registers more efficiently.

Inlining also enables more opportunities for superscalar optimizations [1]. Modern processors run at several times the speed of memory, and loading a value from memory can be the slowest instruction. To get around this problem, the load instruction does not wait for the fetch from memory to complete before allowing the next instruction to execute. The processor will execute instructions following the load while the memory system produces the requested value in parallel. If any instruction attempts to use the register that the memory was fetched into before the load completes, the processor will stall waiting on the load to finish. On the other hand, if the load finishes before any instruction attempts to use the register, then the machine never slows down. To maximize the speed of the machine, compilers attempt to move load instructions to earlier points in the program. A compiler cannot move the first loads in a function to before the function is called, but it can move the first loads in an inline substitution into code before the inlined call.

Inline Disadvantages

The primary disadvantage to inline substitution is that it usually makes the program code bigger. In extreme cases, this can degrade program performance by increasing page faults and cache misses. Reading a page from the disk may take as long as executing hundreds of thousands of instructions. Poor cache performance may slow down a program by a factor of two. Reasonable care must be taken when inlining not to make the program so big that either paging or caching problems dominate the execution time.

There are also functions that it does not pay to inline. Consider:

```
if ((ptr = malloc(100)) == NULL)
    die();
```

where the **die()** function prints an error message, performs a little cleanup, and then exits the program. There might be lots of calls to **die()**, **die()** might even be a very short function, but it would never pay to inline the function since it is never executed. Since the function calls are not executed, you want the calls to be as short as possible in order to minimize page faults and cache misses.

inline Keyword

C99 has added a new keyword, **inline**, which allows the programmer to hint that calls to that function should be inlined. The **inline** keyword may appear anywhere among the storage class specifiers, type specifiers, or type modifiers at the start of a declaration of a function. Some examples:

```
inline float cube(float x) {return
                            x*x*x;}
static int inline h();
inline extern void g();
```

Either **static** or **extern** functions may be declared inline. Unlike C++, a function declared inline without a storage-class specifier is an **extern** function, not a **static** function (more on this later.) Either a function definition or a function prototype may be declared inline. If a function prototype is declared inline, a separate definition of the function must appear somewhere else in the module if the function is called or if the function is **extern**.

Like **register**, the **inline** keyword is only a suggestion that an optimization be performed. Some compilers might ignore it completely and never inline. Others might ignore it and inline based on criteria that usually result in best performance. Still other compilers might only honor the keyword if additional requirements are met by the program.

Inlining a function call is an optimization that a compiler may perform on any call at any time. About the only requirement from the compiler's point of view is that the compiler needs a copy of the body of the function if it is to inline a call to it. Since the optimization produces an identical result as a normal call to the function, compilers do not need any special permission to perform the optimization. In fact, for a number of years now, most compilers do inline substitution as a normal optimization. Therefore, you might find it surprising that C99 added the **inline** keyword. There are three reasons for this.

First, while most compilers have the modern organization described at the start of this article and attempt to do some inlining automatically, there are still some compilers that are written to minimize memory use during compilation or do not attempt any automatic inlining. These compilers benefit from having an inlining hint from the programmer. For example, a small memory footprint compiler might compile a source file a function at a time and normally discard its internal representation of a function being complied after generating code for that function. The **inline** keyword can inform such a compiler to save its internal representation of the function so that it can inline it later. Such compilers might only honor **inline** for calls that appear after the definition (body) of an **inline** function is seen. (Most modern compilers do not have any ordering requirements.)

Second, since inlining has a potential downside, compilers try to be reasonable in making decisions about which functions to inline. The programmer might determine

that inlining is useful for a large function that the compiler would not automatically inline. Some compilers might honor an explicit inline request from the programmer for such functions.

Third, compilers need help from programmers to handle **extern inline** functions because of limitations due to linkers and separate compilation. Unlike normal **extern** functions where the definition (body) of the function appears in only one module, **extern inline** functions need their definitions duplicated in every module that contains calls to the function if those calls are to be inlined. Normally this is done by putting the function definition in a header file and including it where needed so that you only have to maintain a single textual copy of the function. The ramifications of this make up the rest of this column.

extern inline

The duplication of the **extern inline** function causes problems for the compiler. Under some circumstances, the compiler needs to produce a real, callable copy of an **inline** function. This might happen because some of the calls were not inlined, because the address of the function was taken so that it could be called through a pointer, or because the **inline** function was recursive. (Even if a compiler inlines a recursive function in itself a few times, at some point the compiler must generate a real call to the function or compile forever). The problem is how to pick an object file to contain the code for the callable function, which must have a real external name for the linker and a unique address in the program. C99 and C++ have solved this problem differently.

C++ requires that the C++ implementation find some way to automatically pick. In the long term, this solution is probably the right one since it is convenient for the programmer, and eventually the tools used to build programs will handle this gracefully. But, currently this approach has some rough edges. Some C++ compilers solve this problem by always generating a real callable copy of the function in every module that contains a copy of the definition. The linker is modified to throw away silently all but one copy of the function code. The disadvantages of this approach are that it slows down every compilation to produce the callable copies of the functions, it makes object files larger with the redundant copies, and it slows down the linker who must read and discard the extra copies. Other C++ compilers generate a callable copy in the first module ever compiled that contains a definition of the function. The compiler must then maintain a database to be consulted when compiling every module that tells of compilation decisions made in all of the other modules. The contents of an object file depend not only on the source code of the module, but also which other modules have been compiled in the order they were compiled. If the same module is part of two different programs, you may need two different object files for the same module in order to record different decisions about who is responsible for the callable version of an **extern inline** function.

C99 took an alternative approach: it requires the programmer to pick a module to contain the callable copy of an **extern inline** function. By default in C99, **inline** functions without any storage class are **extern**. If all of the declarations of an **inline** function in a module lack a storage class specifier, then the function is an **extern inline** function, and the module will not produce a callable copy of the function. On the other hand, if one declaration of the **inline** function in the module explicitly contains the keyword **extern**, then that module will produce a callable copy of the function.

This leads to the following source code organization for C99. Put definitions of **extern inline** functions in header files and do not use the keyword **extern**. For example, **mymath.h** might include:

```
// mymath.h
inline float square(float x) {return x*x;}
inline float cube(float x) {return x*x*x;}
```

That header can be included in as many modules as you wish. In exactly one module, you should include the header file and then declare prototypes for the functions using the **extern** keyword in order to get callable copies of the functions (those prototype need not repeat the **inline** keyword):

```
// mymath.c
extern float square(float x);
extern float cube(float x);
```

C99 places a few restrictions on **extern inline** functions (**static inline** functions have no restrictions). Because the body of an **extern inline** function will appear in many different modules, an **extern inline** function may not reference **static** functions or objects from the surrounding scope since such objects would be different in every module:

```
static int x;
static void f();
inline void g()
{
  x = 0;  // invalid
  f();    // invalid
}
```

C99 also prohibits **extern inline** functions from declaring **static** objects unless they are not modifiable:

```
inline void h()
{
static int x; // bad
static const float pi=3.1; //ok
}
```

Summary

Inline substitution is a general optimization that can be controlled to some extent using the C99 **inline** keyword. Inlining a function produces the same results as a normal call to the function, but may run faster and may permit more optimization than a normal call. Static **inline** functions have no special considerations, but **extern inline** functions require that the programmer pick one module to contain a real callable version of the function and follow some restrictions about accessing **static**s.

Reference

[1] Randy Meyers. “The New C: Restricted Pointers,” *C/C++ Users Journal*, November 2000.

Randy Meyers *is a consultant providing training and mentoring in C, C++, and Java. He is the current chair of J11, the ANSI C committee, and previously was a member of J16 (ANSI C++) and the ISO Java Study Group. He worked on compilers for Digital Equipment Corporation for 16 years and was Project Architect for DEC C and C++. He can be reached at* ***rmeyers@ix.netcom.com****.*